Learning Dynamics with Synchronous, Asynchronous and General Semantics

Tony Ribeiro, Maxime Folschette, Morgan Magnin, Olivier Roux, Katsumi Inoue

Learning Dynamics with Synchronous, Asynchronous and General Semantics

Tony Ribeiro[1], Maxime Folschette[2], Morgan Magnin[1,3], Olivier Roux[1], and Katsumi Inoue[3]

[1] Laboratoire des Sciences du Numérique de Nantes, 1 rue de la Noë, 44321 Nantes, France
tony_ribeiro@ls2n.fr,
[2] Univ Rennes, Inria, CNRS, IRISA, IRSET, F-35000 Rennes, France
[3] National Institute of Informatics, Tokyo, Japan

Abstract. Learning from interpretation transition *(LFIT)* automatically constructs a model of the dynamics of a system from the observation of its state transitions. So far, the systems that *LFIT* handles are restricted to synchronous deterministic dynamics, i.e., all variables update their values at the same time and, for each state of the system, there is only one possible next state. However, other dynamics exist in the field of logical modeling, in particular the asynchronous semantics which is widely used to model biological systems. In this paper, we focus on a method that learns the dynamics of the system independently of its semantics. For this purpose, we propose a modeling of multi-valued systems as logic programs in which a rule represents what can occurs rather than what will occurs. This modeling allows us to represent non-determinism and to propose an extension of *LFIT* in the form of a semantics free algorithm to learn from discrete multi-valued transitions, regardless of their update schemes. We show through theoretical results that synchronous, asynchronous and general semantics are all captured by this method. Practical evaluation is performed on randomly generated systems and benchmarks from biological literature to study the scalability of this new algorithm regarding the three aforementioned semantics.

Keywords: Dynamical semantics, learning from interpretation transition, dynamical systems, Inductive Logic Programming

1 Introduction

Learning the dynamics of systems with many interactive components becomes more and more important due to many applications, e.g., multi-agent systems, robotics and bioinformatics. Knowledge of a system dynamics can be used by agents and robots for planning and scheduling. In bioinformatics, learning the dynamics of biological systems can correspond to the identification of the influence of genes and can help to understand their interactions. While building a model, the choice of a relevant semantics associated to the studied system represents a major issue with regard to the kind of dynamical properties to

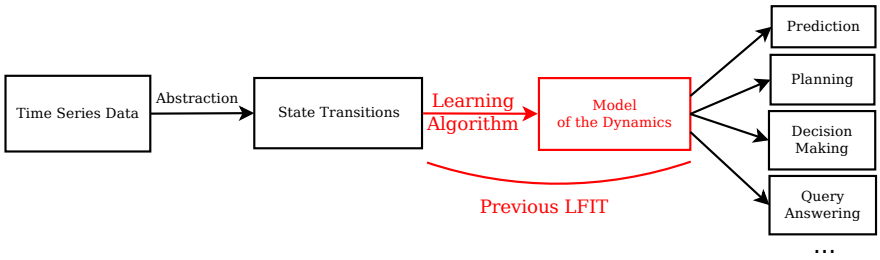

Fig. 1: Assuming a discretization of time series data of a system as state transitions, we propose a method to automatically model the system dynamics.

analyze. The differences and common features of different semantics w.r.t. properties of interest (attractors, oscillators, etc.) constitutes an area of research per itself, especially in the field of Boolean networks. In [8], the author exhibits the translation from Boolean networks into logic programs and discusses the point attractors in both synchronous and asynchronous semantics. In [6], A. Garg et al. address the differences and complementarity of synchronous and asynchronous semantics to model biological networks and identify attractors. The benefits of the synchronous model are to be computationally tractable, while classical state space exploration algorithms fail on asynchronous ones. For some applications, like the biological ones, asynchronous semantics is said to capture more realistic behaviors: at a given time, a single gene can change its expression level. This results in a potential combinatorial explosion of the number of reachable states. To illustrate this issue, the authors of [6] compare the time needed to compute the attractors of various models (mammalian cell, T-helper, dendritic cell, ...) and compare the results with synchronous and asynchronous semantics. More recently, in [3], the authors question the kind of properties that may be preserved, whatever the semantics, while discussing the merits of the usual updating modes including synchronous, fully asynchronous and generalized asynchronous updating. As a good choice of semantics is key to a sound analysis of a system, it is critical to be able to learn not only one kind of semantics, but to embrace a wide range of updating modes.

So far, learning from interpretation transition *(LFIT)* [9] has been proposed to automatically construct a model of the dynamics of a system from the observation of its state transitions. Figure 1 shows this learning process. Given some raw data, like time-series data of gene expression, a discretization of those data in the form of state transitions is assumed. From those state transitions, according to the semantics of the system dynamics, different inference algorithms that model the system as a logic program have been proposed. The semantics of system dynamics can indeed differ with regard to the synchronism of its variables, the determinism of its evolution and the influence of its history. The LFIT framework proposes several modeling and learning algorithms to tackle those different semantics. To date, the following systems have been tackled: memory-less consistent systems [9], systems with memory [14], non-consistent systems [12]

and their multi-valued extensions [15,11]. All those methods are dedicated to discrete systems or assume an abstraction of time series data as discrete transitions. [16] proposes a method that allows to deal with continuous time series data, the abstraction itself being learned by the algorithm.

As a summary, the systems that *LFIT* handles so far are restricted to synchronous deterministic dynamics, i.e., all variables update their values at the same time and, for each state of the system, there is only one possible next state. However, as we said previously, other dynamics exist in the field of logical modeling, in particular the asynchronous and generalized semantics which are of deep interest to model biological systems.

In this paper, we focus on a method that learns the dynamics of the system independently of its dynamics semantics. For this purpose, we propose a modeling of discrete multi-valued systems as logic programs in which each rule represents that a variable possibly takes some value at the next state, extending the formalism introduced in [12,15,11]. Research in multi-valued logic programming has proceeded along three different directions [10]: bilattice-based logics [5,7], quantitative rule sets [17] and annotated logics [2,1]. The multi-valued logic representation used in our new algorithm is based on annotated logics. Here, to each variable corresponds a domain of discrete values. In a rule, a literal is an atom annotated with one of these values. It allows us to represent annotated atoms simply as classical atoms and thus to remain in the normal logic program semantics. This modeling allows us to represent non-determinism and to propose an extension of *LFIT* in the form of a semantics free algorithm to learn from discrete multi-valued transitions, regardless of their update schemes. We show from theoretical results and experimental evaluation that our new algorithm can learn systems dynamics from both synchronous (deterministic or not), asynchronous and general semantics transitions.

The organization of the paper is as follows. Section 2 provides a formalization of multi-valued logic program, dynamics semantics under logic programs, the learning operations and their properties. Section 3 presents the **GULA** learning algorithm and Section 4 its experimental evaluation. Section 5 concludes the paper and provides possible outlooks about applications and improvements of the method. All proofs of theorems and propositions are given in Appendix.

2 Formalization

In this section, the concepts necessary to understand the learning algorithm are formalized. In Sect. 2.1 the basic notions of *multi-valued logic* ($\mathcal{M}$VL) and a number of important properties that the learned programs must have are presented. Then in Sect. 2.2 the operations that are performed during the learning, as well as results about the preservation of the properties introduced in Sect. 2.1 throughout the learning are exposed. Finally, Sect. 2.3 introduces the formalization of several dynamical semantics of multi-valued logic and show that the learning process of Sect. 2.2 is independent of the chosen semantics.

In the following, we denote by $\mathbb{N} := \{0, 1, 2, ...\}$ the set of natural numbers, and for all $k, n \in \mathbb{N}$, $[\![k; n]\!] := \{i \in \mathbb{N} \mid k \leq i \leq n\}$ is the set of natural numbers between k and n included. For any set S, the cardinal of S is denoted $|S|$.

2.1 Multi-valued Logic Program

Let $\mathcal{V} = \{\mathrm{v}_1, \ldots, \mathrm{v}_n\}$ be a finite set of $n \in \mathbb{N}$ variables, and $\mathsf{dom} : \mathcal{V} \to \mathbb{N}$ a function associating a maximum value (and thus a domain) to each variable. The atoms of $\mathcal{M}$VL are of the form v^{val} where $\mathrm{v} \in \mathcal{V}$ and $val \in [\![0; \mathsf{dom}(\mathrm{v})]\!]$. The set of such atoms is denoted by $\mathcal{A}_{\mathsf{dom}}^{\mathcal{V}}$ for a given set of variables $\mathcal{V}$ and a given domain function dom. In the following, we work on specific $\mathcal{V}$ and dom that we omit to mention when the context makes no ambiguity, thus simply writing $\mathcal{A}$.

A $\mathcal{M}$VL rule R is defined by:

$$R \quad = \quad \mathrm{v}_0^{val_0} \leftarrow \mathrm{v}_1^{val_1} \wedge \cdots \wedge \mathrm{v}_m^{val_m} \tag{1}$$

where $\forall i \in [\![0; m]\!], \mathrm{v}_i^{val_i} \in \mathcal{A}$ are atoms in $\mathcal{M}$VL so that every variable is mentioned at most once in the right-hand part: $\forall j, k \in [\![1; m]\!], j \neq k \Rightarrow \mathrm{v}_j \neq \mathrm{v}_k$. Intuitively, the rule R has the following meaning: the variable v_0 can take the value val_0 in the next dynamical step if for each $i \in [\![1; m]\!]$, variable v_i has value val_i in the current step.

The atom on the left-hand side of the arrow is called the *head* of R and is denoted $h(R) := \mathrm{v}_0^{val_0}$. The notation $\mathrm{var}(h(R)) := \mathrm{v}_0$ denotes the variable that occurs in $h(R)$. The conjunction on the right-hand side of the arrow is called the *body* of R, written $b(R)$ and can be assimilated to the set $\{\mathrm{v}_1^{val_1}, \ldots, \mathrm{v}_m^{val_m}\}$; we thus use set operations such as $\in$ and $\cap$ on it. A *multi-valued logic program* ($\mathcal{M}$VLP) is a set of $\mathcal{M}$VL rules.

In the following, we define several notions on $\mathcal{M}$VL rules and programs that will be used for dynamics learning. Def. 1 introduces a domination relation between rules that defines a partial anti-symmetric ordering, as stated by Proposition 1.

Definition 1 (Rule Domination). *Let R_1, R_2 be two $\mathcal{M}$VL rules. The rule R_1* dominates *R_2, written $R_2 \leq R_1$ if $h(R_1) = h(R_2)$ and $b(R_1) \subseteq b(R_2)$.*

Proposition 1. *Let R_1, R_2 be two $\mathcal{M}$VL rules. If $R_1 \leq R_2$ and $R_2 \leq R_1$ then $R_1 = R_2$.*

Rules with the most general bodies dominate the other rules. In practice, these are the rules we are interested in since they cover the most general cases.

The dynamical system we want to learn the rules of is represented by a succession of *states* as formally given by Def. 2. We also define the compatibility of a rule with a state in Def. 3 and with another rule in Def. 4, and give a property on this last notions in Proposition 2.

Definition 2 (Discrete state). *A* discrete state *s is a function from $\mathcal{V}$ to $\mathbb{N}$, i.e., it associates an integer value to each variable in $\mathcal{V}$. It can be equivalently*

represented by the set of atoms $\{\mathrm{v}^{s(\mathrm{v})} \mid \mathrm{v} \in \mathcal{V}\}$ *and thus we can use classical set operations on it. We write* $\mathcal{S}$ *to denote the set of all discrete states, and a couple of states* $(s, s') \in \mathcal{S}^2$ *is called a* transition*.*

Definition 3 (Rule-state matching). *Let* $s \in \mathcal{S}$*. The* $\mathcal{M}$VL *rule* R matches s*, written* $R \sqcap s$*, if* $b(R) \subseteq s$*.*

Definition 4 (Cross-matching). *Let* R *and* R' *be two* $\mathcal{M}$VL *rules. These rules* cross-match*, written* $R \sqcap R'$ *when there exists* $s \in \mathcal{S}$ *such that* $R \sqcap s$ *and* $R' \sqcap s$*.*

Proposition 2 (Cross-matching). *Let* R *and* R' *be two* $\mathcal{M}$VL *rules.*

$$R \sqcap R' \text{ iff } \forall \mathrm{v} \in \mathcal{V}, \forall val, val' \in \mathbb{N}, (\mathrm{v}^{val}, \mathrm{v}^{val'}) \in b(R) \times b(R') \implies val = val'.$$

The final program we want to learn should be complete and consistent within itself and with the observed transitions. The following definitions formalize these desired properties. In Def. 5 we characterize the fact that a rule of a program is useful to describe the dynamics of one variable in a transition; this notion is then extended to a program and a set of transitions, under the condition that there exists such a rule for each variable and each transition. A conflict (Def. 6) arises when a rule describes a change that is not featured in the considered set of transitions. Two rules are concurrent (Def. 7) if they cross-match but have a different head on the same variable. Finally, Def. 8 and Def. 9 give the characteristics of a complete (the whole dynamics is covered) and consistent (without conflict) program.

Definition 5 (Rule and program realization). *Let* R *be a* $\mathcal{M}$VL *rule and* $(s, s') \in \mathcal{S}^2$*. The rule* R realizes *the transition* (s, s')*, written* $s \xrightarrow{R} s'$*, if* $R \sqcap s \wedge h(R) \in s'$*.*

A $\mathcal{M}$VLP P realizes $(s, s') \in \mathcal{S}^2$*, written* $s \xrightarrow{P} s'$*, if* $\forall \mathrm{v} \in \mathcal{V}, \exists R \in P, \mathrm{var}(h(R)) = \mathrm{v} \wedge s \xrightarrow{R} s'$*. It* realizes $T \subseteq \mathcal{S}^2$*, written* $\xhookrightarrow{P} T$*, if* $\forall (s, s') \in T, s \xrightarrow{P} s'$*.*

In the following, for all sets of transitions $T \subseteq \mathcal{S}^2$, we denote: $\mathrm{fst}(T) := \{s \in \mathcal{S} \mid \exists (s_1, s_2) \in T, s_1 = s\}$. We note that $\mathrm{fst}(T) = \emptyset \implies T = \emptyset$.

Definition 6 (Conflicts). *A* $\mathcal{M}$VL *rule* R conflicts *with a set of transitions* $T \subseteq \mathcal{S}^2$ *when* $\exists s \in \mathrm{fst}(T), \big(R \sqcap s \wedge \forall (s, s') \in T, h(R) \notin s'\big)$*.*

Definition 7 (Concurrent rules). *Two* $\mathcal{M}$VL *rules* R *and* R' *are* concurrent *when* $R \sqcap R' \wedge \mathrm{var}(h(R)) = \mathrm{var}(h(R')) \wedge h(R) \neq h(R')$*.*

Definition 8 (Complete program). *A* $\mathcal{M}$VLP P *is* complete *if* $\forall s \in \mathcal{S}, \forall \mathrm{v} \in \mathcal{V}, \exists R \in P, R \sqcap s \wedge \mathrm{var}(h(R)) = \mathrm{v}$*.*

Definition 9 (Consistent program). *A* $\mathcal{M}$VLP P *is* consistent *with a set of transitions* T *if* P *does not contains any rule* R *conflicting with* T*.*

2.2 Learning operations

This section focuses on the manipulation of programs for the learning process. Def. 10 and Def. 11 formalize the main atomic operations performed on a rule or a program by the learning algorithm, whose objective is to make minimal modifications to a given $\mathcal{M}$VLP in order to be consistent with a new set of transitions.

Definition 10 (Rule least specialization). *Let R be a $\mathcal{M}$VL rule and $s \in \mathcal{S}$ such that $R \sqcap s$. The* least specialization *of R by s is:*

$$L_{\text{spe}}(R, s) := \{h(R) \leftarrow b(R) \cup \{\mathrm{v}^{val}\} \mid \mathrm{v}^{val} \in \mathcal{A} \wedge \mathrm{v}^{val} \notin s \wedge \forall val' \in \mathbb{N}, \mathrm{v}^{val'} \notin b(R)\}.$$

Definition 11 (Program least revision). *Let P be a $\mathcal{M}$VLP, $s \in \mathcal{S}$ and $T \subseteq \mathcal{S}^2$ such that $\text{fst}(T) = \{s\}$. Let $R_P := \{R \in P \mid R$ conflicts with $T\}$. The* least revision *of P by T is $L_{\text{rev}}(P, T) := (P \setminus R_P) \cup \bigcup_{R \in R_P} L_{\text{spe}}(R, s)$.*

Theorem 1 states properties on the least revision, in order to prove it suitable to be used in the learning algorithm.

Theorem 1. *Let R be a $\mathcal{M}$VL rule and $s \in \mathcal{S}$ such that $R \sqcap s$. Let $S_R := \{s' \in \mathcal{S} \mid R \sqcap s'\}$ and $S_{\text{spe}} := \{s' \in \mathcal{S} \mid \exists R' \in L_{\text{spe}}(R, s), R' \sqcap s'\}$.*

Let P be a $\mathcal{M}$VLP and $T, T' \subseteq \mathcal{S}^2$ such that $|\text{fst}(T)| = 1 \wedge \text{fst}(T) \cap \text{fst}(T') = \emptyset$. The following results hold:

1. $S_{\text{spe}} = S_R \setminus \{s\}$,
2. $L_{\text{rev}}(P, T)$ *is consistent with* T,
3. $\overset{P}{\hookrightarrow} T' \implies \overset{L_{\text{rev}}(P,T)}{\hookrightarrow} T'$,
4. $\overset{P}{\hookrightarrow} T \implies \overset{L_{\text{rev}}(P,T)}{\hookrightarrow} T$,
5. P *is complete* $\implies L_{\text{rev}}(P, T)$ *is complete.*

Proof sketch. The first two points follow from Def. 10 and 11. The third point follows from Def. 5 and the first point. The fourth point follows from Def. 5 and 11. The last point follows from Def. 8 and the first point. □

Def. 12 groups all the properties that we want the learned program to have: suitability and optimality, and Proposition 3 states that the optimal program of a set of transitions is unique.

Definition 12 (Suitable and optimal program). *Let $T \subseteq \mathcal{S}^2$. A $\mathcal{M}$VLP P* is suitable for T *when:*

- *P is consistent with T,*
- *P realizes T,*
- *P is complete*
- *for all $\mathcal{M}$VL rules R not conflicting with T, there exists $R' \in P$ such that $R \leq R'$.*

If in addition, for all $R \in P$, all the $\mathcal{M}$VL rules R' belonging to $\mathcal{M}$VLP suitable for T are such that $R \leq R'$ implies $R' \leq R$ then P is called optimal.

Proposition 3. *Let $T \subseteq \mathcal{S}^2$. The $\mathcal{M}$*VLP *optimal for T is unique and denoted $P_{\mathcal{O}}(T)$.*

Proof sketch. Reasoning by contradiction, a rule that should occur in only one $\mathcal{M}$VLP optimal for T necessarily occurs in another one. □

The next properties are directly used in the learning algorithm. Proposition 4 gives an explicit definition of the optimal program for an empty set of transitions, which is the starting point of the algorithm. Proposition 5 gives a method to obtain the optimal program from any suitable program by simply removing the dominated rules; this means that the $\mathcal{M}$VLP optimal for a set of transitions can be obtained from any $\mathcal{M}$VLP suitable for the same set of transitions by removing all the dominated rules. Finally, in association with these two results, Theorem 2 gives a method to iteratively compute $P_{\mathcal{O}}(T)$ for any $T \subseteq \mathcal{S}^2$, starting from $P_{\mathcal{O}}(\emptyset)$.

Proposition 4. $P_{\mathcal{O}}(\emptyset) = \{\mathrm{v}^{val} \leftarrow \emptyset \mid \mathrm{v}^{val} \in \mathcal{A}\}$.

Proof sketch. By construction. □

Proposition 5. *Let $T \subseteq \mathcal{S}^2$. If P is a $\mathcal{M}$*VLP *suitable for T, then $P_{\mathcal{O}}(T) = \{R \in P \mid \forall R' \in P, R \leq R' \implies R' \leq R\}$*

Theorem 2. *Let $s \in \mathcal{S}$ and $T, T' \subseteq \mathcal{S}^2$ such that $|\mathrm{fst}(T')| = 1 \wedge \mathrm{fst}(T) \cap \mathrm{fst}(T') = \emptyset$. $L_{\mathrm{rev}}(P_{\mathcal{O}}(T), T')$ is a $\mathcal{M}$*VLP *suitable for $T \cup T'$.*

Proof sketch. Consistency is proved by contradiction. Completeness and realization stem from Theorem 1. The final point is proved by exhibiting for each R not conflicting with $T' \cup T$ the rule in $L_{\mathrm{spe}}(P_{\mathcal{O}}(T'), T)$ that dominates it. □

2.3 Dynamical semantics

In this section, we first formalize the notion of semantics, in Def. 13, as an update policy based on a program. More formally, a semantics is a function that, to a complete program, associates a set of transitions where each state has at least one outgoing transition. Such a set of transitions can also be seen as a function that maps any state to a non-empty set of states, regarded as possible dynamical branchings. We give examples of semantics afterwards.

Definition 13 (Semantics). *Let $\mathcal{A}^{\mathcal{V}}_{\mathsf{dom}}$ be a set of atoms and $\mathcal{S}$ the corresponding set of states. A* semantics *(on $\mathcal{A}^{\mathcal{V}}_{\mathsf{dom}}$) is a function that associates, to each complete $\mathcal{M}$*VLP *P, a set of transitions $T \subseteq \mathcal{S}^2$ so that:* $\mathrm{fst}(T) = \mathcal{S}$. *Equivalently, a semantics can be seen as a function of* $\big(\text{c-}\mathcal{M}\text{VLP} \to (\mathcal{S} \to \wp(\mathcal{S}) \setminus \emptyset)\big)$ *where* c-$\mathcal{M}$VLP *is the set of complete $\mathcal{M}$*VLP*s and $\wp$ is the power set operator.*

In the following, we present a formal definition and a characterization of three particular semantics that are widespread in the field of complex dynamical systems: synchronous, asynchronous and general, and we also treat the particular case of the deterministic synchronous semantics. Note that some points in

these definitions are arbitrary and could be discussed depending on the modeling paradigm. For instance, the policy about rules R so that $\exists s \in \mathcal{S}, s \sqcap R \wedge h(R) \in s$, which model stability in the dynamics, could be to include them (such as in the synchronous and general semantics) or exclude them (such as in the asynchronous semantics) from the possible dynamics. The learning method of this paper is independent to the considered semantics as long as it respects Def. 13.

Def. 14 introduces the synchronous semantics, consisting in updating each variables with one rule at each step, in order to compute the next state. However, this is taken in a loose sense: as stated above, rules that make a variable change its value are not prioritized over rules that don't. Furthermore, if several rules on a same variable match the current state, then several transitions are possible, depending on which rule is applied. Thus, for a self-transition (s, s) to occur, there needs to be, for each atom $\mathrm{v}^{val} \in s$, a rule that matches s and whose head is v^{val}. Note however that such a loop is not necessarily a point attractor; it is only the case if all rules of P that match s have their head in s.

Definition 14 (Synchronous semantics). *The* synchronous semantics $\mathcal{T}_{syn}$ *is defined by:*

$$\mathcal{T}_{syn} : P \mapsto \{(s, s') \in \mathcal{S}^2 \mid s' \subseteq \{h(R) \in \mathcal{A} \mid R \in P, R \sqcap s\}\}$$

We note that if two different transitions are observed from the same state s, all states that are combinations of those two states are also successors of s. This is used in Proposition 6 as a characterization of the synchronous semantics.

In the following, if $s \in \mathcal{S}$ is a state and $X \subseteq \mathcal{A}$ is a set of atoms such that $\forall \mathrm{v}_1^{val_1}, \mathrm{v}_2^{val_2} \in X, \mathrm{v}_1 = \mathrm{v}_2 \implies val_1 = val_2$, we denote: $s \backslash\backslash X := \{\mathrm{v}^{val} \in s \mid \mathrm{v} \notin \{\mathrm{w} \mid \mathrm{w}^{val'} \in X\}\} \cup X$. In other words, $s \backslash\backslash X$ is the discrete state s where all variables mentioned in X have their value replaced by the value in X.

Proposition 6 (Synchronous transitions). *Let* $T \subseteq \mathcal{S}^2$ *so that* $\mathrm{fst}(T) = \mathcal{S}$. *The transitions of* T *are* synchronous*, i.e.,* $\exists P$ *a* $\mathcal{M}$VLP *such that* $\mathcal{T}_{syn}(P) = T$, *if and only if* $\forall (s, s_1), (s, s_2) \in T, \forall s_3 \in \mathcal{S}, s_3 \subseteq s_1 \cup s_2 \implies (s, s_3) \in T$.

Proof sketch. ($\Rightarrow$) By definition of $\mathcal{T}_{syn}$. ($\Leftarrow$) Consider the most naive program P realizing T; the characterization applied iteratively allows to conclude that $\mathcal{T}_{syn}(P) \subseteq T$ while $T \subseteq \mathcal{T}_{syn}(P)$ comes by definition of P and $\mathcal{T}_{syn}$. □

In Def. 15, we formalize the asynchronous semantics that imposes that no more than one variable can change its value in each transition. Contrary to the previous one, this semantics prioritizes the changes. Thus, for a self-transition (s, s) to occur, it is required that all rules of P that match s have their head in s, i.e., this only happens when (s, s) is a point attractor. Proposition 7 characterizes the asynchronous semantics by stating that from a state s, either the only successor is s, or all successors differ from s by exactly one atom.

Definition 15 (Asynchronous semantics). *The* asynchronous semantics $\mathcal{T}_{asyn}$ *is defined by:*

$$\begin{aligned}\mathcal{T}_{asyn} : P \mapsto \{(s, s \backslash\backslash \{h(R)\}) \in \mathcal{S}^2 \mid R \in P \wedge R \sqcap s \wedge h(R) \notin s\} \\ \cup \{(s, s) \in \mathcal{S}^2 \mid \forall R \in P, R \sqcap s \implies h(R) \in s\}.\end{aligned}$$

Proposition 7 (Asynchronous transitions). *Let $T \subseteq \mathcal{S}^2$ so that* $\mathrm{fst}(T) = \mathcal{S}$. *The transitions of T are* asynchronous*, i.e., $\exists P$ a $\mathcal{M}$VLP such that $\mathcal{T}_{asyn}(P) = T$, if and only if $\forall s, s' \in \mathcal{S}, s \neq s', \big((s,s) \in T \implies (s,s') \notin T\big) \wedge \big((s,s') \in T \implies |s \setminus s'| = 1\big)$.*

Proof sketch. When handling a transition (s, s'), consider separately the cases $s = s'$ and $s \neq s'$. ($\Rightarrow$) By contradiction, based the definition of $\mathcal{T}_{asyn}$. ($\Leftarrow$) Consider the most naive program P realizing T; from the characterization, it comes: $\mathcal{T}_{syn}(P) = T$. □

Finally, Def. 16 formalizes the general semantics as a more permissive version of the synchronous one: any subset of the variables can change their value in a transition. A self-transition (s, s) thus occurs for each state s because the empty set of variables can always be selected for update. However, as for the synchronous semantics, such a self-transition is a point attractor only if all rules of P that match s have their head in s. Proposition 8 is a characterization of the general semantics. It is similar to the synchronous characterization, but the combination of the two successor states is also combined with the origin state.

Definition 16 (General semantics). *The* general semantics $\mathcal{T}_{gen}$ *is defined by:*

$$\mathcal{T}_{gen} : P \mapsto \{(s, s \Vdash r) \in \mathcal{S}^2 \mid r \subseteq \{h(R) \in \mathcal{A} \mid R \in P \wedge R \sqcap s\} \wedge \forall \mathrm{v}_1^{val_1}, \mathrm{v}_2^{val_2} \in r, \mathrm{v}_1 = \mathrm{v}_2 \implies val_1 = val_2\}.$$

Proposition 8 (General transitions). *Let $T \subseteq \mathcal{S}^2$ so that* $\mathrm{fst}(T) = \mathcal{S}$. *The transitions of T are* general*, i.e., $\exists P$ a $\mathcal{M}$VLP such that $\mathcal{T}_{gen}(P) = T$, if and only if: $\forall (s, s_1), (s, s_2) \in T, \forall s_3 \in \mathcal{S}, s_3 \subseteq s \cup s_1 \cup s_2 \implies (s, s_3) \in T$.*

Proof sketch. Similar to the synchronous case. □

In addition, in Def. 17, we define the notion of deterministic dynamics, that is, a set of transitions with no "branching", and give a particular characterization of deterministic dynamics in the synchronous case in Proposition 9.

Definition 17 (Deterministic transitions). *A set of transitions $T \subseteq \mathcal{S}^2$ is* deterministic *if $\forall (s, s') \in T, \nexists (s, s'') \in T, s'' \neq s'$. A $\mathcal{M}$VLP P is* deterministic regarding a semantics *if the set of all transitions T_P obtained by applying the semantics on P is deterministic.*

Proposition 9 (Synchronous Deterministic program). *A $\mathcal{M}$VLP P produces* deterministic *transitions with synchronous semantics if it does not contain concurrent rules, i.e., $\forall R, R' \in P, \big(\mathrm{var}(h(R)) = \mathrm{var}(h(R')) \wedge R \sqcap R'\big) \implies h(R) = h(R')$.*

Until now, the LFIT algorithm only tackled the learning of synchronous deterministic program. Using the formalism introduced in the previous sections, it can now be extended to learn systems from transitions produced from the three semantics defined above. Furthermore, both deterministic and non-deterministic systems can now be learned.

Finally, with Theorem 3, we state that the definitions and method developed in the previous section are independent of the chosen semantics.

Theorem 3 (Semantics-free correctness). *Let P be a $\mathcal{M}$VLP such that P is complete.*

- $\mathcal{T}_{syn}(P) = \mathcal{T}_{syn}(P_{\mathcal{O}}(\mathcal{T}_{syn}(P)))$,
- $\mathcal{T}_{asyn}(P) = \mathcal{T}_{asyn}(P_{\mathcal{O}}(\mathcal{T}_{asyn}(P)))$,
- $\mathcal{T}_{gen}(P) = \mathcal{T}_{gen}(P_{\mathcal{O}}(\mathcal{T}_{gen}(P)))$.

Proof sketch. Using the properties of an optimal program (Def. 12) and by contradiction. □

3 GULA

In this section we present **GULA**: the General Usage LFIT Algorithm, an extension of the **LF1T** algorithm to capture both synchronous, asynchronous and general semantics dynamics. **GULA** learns a logic program from the observations of its state transitions. Given as input a set of transitions T, **GULA** iteratively constructs a model of the system by applying the method formalized in the previous section as follows:

GULA:

- **INPUT:** a set of atoms $\mathcal{A}$ and a set and of transitions $T \subseteq \mathcal{S}^2$.
- For each atom $\mathrm{v}^{val} \in \mathcal{A}$
  - Extract all states from which no transition to v^{val} exist: $Neg_{\mathrm{v}^{val}} := \{s \mid \nexists (s, s') \in T, \mathrm{v}^{val} \in s'\}$
  - Initialize $P_{\mathrm{v}^{val}} := \{\mathrm{v}^{val} \leftarrow \emptyset\}$
  - For each state $s \in Neg_{\mathrm{v}^{val}}$
    - Extract each rule R of $P_{\mathrm{v}^{val}}$ that matches s: $M_{\mathrm{v}^{val}} := \{R \in P \mid b(R) \subseteq s\}, P_{\mathrm{v}^{val}} := P_{\mathrm{v}^{val}} \setminus M_{\mathrm{v}^{val}}$.
    - For each rule $R \in M_{\mathrm{v}^{val}}$
      - Compute its least specialization $P' = L_{\mathrm{spe}}(R, s)$.
      - Remove all the rules in P' dominated by a rule in $P_{\mathrm{v}^{val}}$.
      - Remove all the rules in $P_{\mathrm{v}^{val}}$ dominated by a rule in P'.
      - Add all remaining rules in P' to $P_{\mathrm{v}^{val}}$.
  - $P := P \cup P_{\mathrm{v}^{val}}$
- **OUTPUT:** $P_{\mathcal{O}}(T) := P$.

Algorithms 1 and 2 provide the detailed pseudocode of the algorithm. Algorithm 1 learns from a set of transitions T the conditions under which each value *val* of each variable v may appear in the next state. Here, learning is performed iteratively for each value of variable to keep the pseudo-code simple. But the process can easily be parallelized by running each loop in an independent thread, bounding the run time to the variable for which the learning is the longest. The algorithm starts by the pre-processing of the input transitions. Lines 4–13 of Algorithm 1 correspond to the extraction of $Neg_{\mathrm{v}^{val}}$, the set of all negative examples of the appearance of v^{val} in next state: all states such that v never takes

the value *val* in the next state of a transition of T. Those negative examples are then used during the following learning phase (lines 14–32) to iteratively learn the set of rules $P_{\mathcal{O}}(T)$. The learning phase starts by initializing a set of rules $P_{\mathrm{v}^{val}}$ to $\{R \in P_{\mathcal{O}}(\emptyset) \mid h(R) = \mathrm{v}^{val}\} = \{\mathrm{v}^{val} \leftarrow \emptyset\}$ (see Def. 12). $P_{\mathrm{v}^{val}}$ is iteratively revised against each negative example *neg* in $Neg_{\mathrm{v}^{val}}$. All rules R_m of $P_{\mathrm{v}^{val}}$ that match *neg* have to be revised. In order for $P_{\mathrm{v}^{val}}$ to remain optimal, the revision of each R_m must not match *neg* but still matches every other state that R_m matches. To ensure that, the least specialization (see Def. 10) is used to revise each conflicting rule R_m. Algorithm 2 shows the pseudo code of this operation. For each variable of $\mathcal{V}$ so that $b(R_m)$ has no condition over it, a condition over another value than the one observed in state *neg* can be added (lines 3–8). None of those revision match *neg* and all states matched by R_m are still matched by at least one of its revision. The revised rules are then added to $P_{\mathrm{v}^{val}}$ after discarding the dominated rules. Once $P_{\mathrm{v}^{val}}$ has been revised against all negatives example of $Neg_{\mathrm{v}^{val}}$, $P = \{R \in P_{\mathcal{O}}(T) \mid h(R) = \mathrm{v}^{val}\}$ and $P_{\mathrm{v}^{val}}$ is added to P. Once all values of each variable have been treated, the algorithm outputs P which becomes $P_{\mathcal{O}}(T)$. Theorem 4 gives good properties of the algorithm, Theorem 5 states that **GULA** can learn from both synchronous, asynchronous and general semantics transitions and Theorem 11 characterizes its time and memory complexities.

Theorem 4 (GULA Termination, soundness, completeness, optimality). *Let $T \subseteq \mathcal{S}^2$. The call* $\mathbf{GULA}(\mathcal{A}, T)$ *terminates and* $\mathbf{GULA}(\mathcal{A}, T) = P_{\mathcal{O}}(T)$.

Theorem 5 (Semantic-freeness). *Let P be a* $\mathcal{M}$VLP *such that P is complete. From Theorem 3 and Theorem 4, the following holds:*

- $\mathbf{GULA}(\mathcal{A}, \mathcal{T}_{syn}(P)) = P_{\mathcal{O}}(\mathcal{T}_{syn}(P))$
- $\mathbf{GULA}(\mathcal{A}, \mathcal{T}_{asyn}(P)) = P_{\mathcal{O}}(\mathcal{T}_{asyn}(P))$
- $\mathbf{GULA}(\mathcal{A}, \mathcal{T}_{gen}(P)) = P_{\mathcal{O}}(\mathcal{T}_{gen}(P))$

Theorem 6 (GULA Complexity). *Let $T \subseteq \mathcal{S}^2$ be a set of transitions, $n := |\mathcal{V}|$ be the number of variables of the system and $d := \max(\mathsf{dom}(\mathcal{V}))$ be the maximal number of values of its variables. The worst-case time complexity of* **GULA** *when learning from T belongs to $\mathcal{O}(|T|^2 + 2n^3d^{2n+1} + 2n^2d^n)$ and its worst-case memory use belongs to $\mathcal{O}(d^{2n} + 2d^n + nd^{n+2})$.*

4 Evaluation

In this section, the benefits from **GULA** are demonstrated on a case study and its scalability is assessed w.r.t. system size and input size, i.e., the number of variables and transitions. All experiments[4] were conducted on an Intel Core I7 (6700, 3.4 GHz) with 32 Gb of RAM. Figure 2 shows results of the evaluations of **GULA** scalability w.r.t. the number of variables of system, the domains size of the variables and the number of transitions in the input of the algorithm.

[4] Available at: `http://tonyribeiro.fr/data/experiments/ILP_2018.zip`

Algorithm 1 GULA($\mathcal{A}$,T)

INPUT: A set of atoms $\mathcal{A}$ and a set of transitions $T \subseteq \mathcal{S}^2$
OUTPUT: $P = P_{\mathcal{O}}(T)$

for each $\mathrm{v}^{val} \in \mathcal{A}$ **do**
  // 1) Extraction of negative examples
  $Neg_{\mathrm{v}val} := \emptyset$
  for each $(s_1, s_1') \in T$ **do**
    $negative\_example := true$
    for each $(s_2, s_2') \in T$ **do**
      if $s_1 == s_2$ and $\mathrm{v}^{val} \in s_2'$ **then**
        $negative\_example := false$
        Break
    if $negative\_example == true$ **then**
      $Neg_{\mathrm{v}val} := Neg_{\mathrm{v}val} \cup \{s_1\}$

  // 2) Revision of the rules of v^{val} to avoid matching of negative examples
  $P_{\mathrm{v}val} := \{\mathrm{v}^{val} \leftarrow \emptyset\}$
  for each $neg \in Neg_{\mathrm{v}val}$ **do**
    $M := \emptyset$
    for each $R \in P_{\mathrm{v}val}$ **do** // Extract all rules that conflict
      if $b(R) \subseteq neg$ **then**
        $M := M \cup \{R\}; P := P \setminus \{R\}$
    for each $R_m \in M$ **do** // Revise each conflicting rule
      $LS := least\_specialization(R_m, neg, \mathcal{A})$
      for each $R_{ls} \in LS$ **do**
        for each $R_p \in P_{\mathrm{v}val}$ **do** // Check if the revision is dominated
          if $b(R_p) \subseteq b(R_{ls})$ **then**
            $dominated := true$
            break
        if $dominated == false$ **then** // Remove old rules that are now dominated
          for each $R_p \in P$ **do**
            if $b(R_{ls}) \subseteq b(R_p)$ **then**
              $P_{\mathrm{v}val} := P_{\mathrm{v}val} \setminus \{R_p\}$
          $P_{\mathrm{v}val} := P_{\mathrm{v}val} \cup \{R_{ls}\}$ // Add the revision
  $P := P \cup P_{\mathrm{v}val}$
return P

For the first experiment, each variable domain is fixed to 3 values and the size of the system evolves from 1 to 20 variables. Run time performances are evaluated on synchronous, asynchronous and general semantics. Given a number of variables and a semantics, the transitions of a random system are directly generated as follows. First all possible states of the system are generated, i.e., $\mathcal{S}$. For each state, a set $S \subseteq \mathcal{A}$ is randomly generated by adding each $\mathrm{v}^{val} \in \mathcal{A}$ with 50% chance, simulating the matching of rules of each variable value. All possible transitions w.r.t. the considered semantics are then generated according to Proposition 6, 7 or 8. A total of 10 such instances are generated and successively given as input to the algorithm. The average run time in seconds (log scale) is given in Figure 2 (top left) w.r.t. to the number of variables of the system learned. In those experiments, the algorithm succeeds to handle systems of at most 8 variables before reaching the time-out of 10,000 seconds.

For the second experiment, the number of variables is fixed to 3 and the domain of each variable evolves from 2 to 10 values. Run time performances are once again evaluated on the three semantics. The instances are generated in the same manner as in previous experiment. The average run time in seconds

Algorithm 2 least_specialization(R, s, $\mathcal{A}$) : specialize R to avoid matching of s

INPUT: a rule R, a state s and a set of atoms $\mathcal{A}$
OUTPUT: a set of rules LS which is the least specialization of R by s according to $\mathcal{A}$.

$LS := \emptyset$
// Revise the rules by least specialization
for each $v^{val} \in s$ **do**
  if $\nexists v^{val'} \in b(R)$ **then** // Add condition for all values not appearing in s
    for each $v^{val''} \in \mathcal{A}, val'' \neq val$ **do**
      $R' := h(R) \leftarrow (b(R) \cup \{v^{val''}\})$
      $LS := LS \cup \{R'\}$
return LS

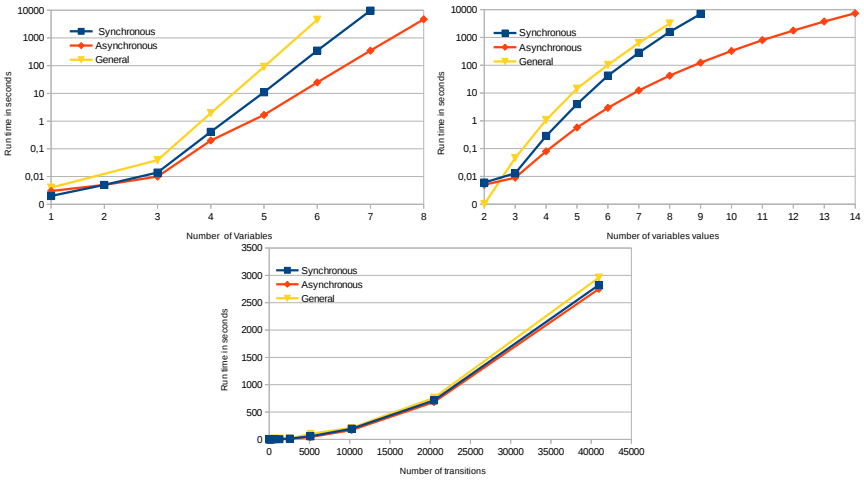

Fig. 2: Evaluation of **GULA**'s scalability w.r.t. number of variables (top left), number of variables values (top right) and number of input transitions (bottom).

(log scale) is given in Figure 2 (top right) w.r.t. to the size of the domains of variables in the system. In those experiment the algorithm succeeds to handle systems with variable of at most 14 values before reaching the time-out of 10,000 seconds.

Finally, for the third experiment, the system size is fixed to 13 variables and we change the number of transitions given as input to the algorithm from 10 to 50,000. Given a number N and a semantics, a set of transitions T is generated in the same manner as previous experiments except that the generation stops when $|T| = N$. Again, a total of 10 such instances are generated and successively given as input to the algorithm, the average run time in second is given in Figure 2 (bottom) w.r.t. the number of input transitions. In those experiment the algorithm succeeds to handle at most 40,000 transitions before reaching the time-out of 10,000 seconds.

The first and second experiment show the exponential impact of the system size on the learning time. The third experiment shows that the run time of the algorithm is rather polynomial w.r.t. to the number of input transitions. The

exponential shape of the first results is explained by the fact that the number of input transitions is exponentially higher when we increase either the number of variables or the variables domains. General semantics instances are longer to learn than the two others, and synchronous semantics instances are longer to learn than asynchronous ones. The same reasoning w.r.t. the number of input transitions holds here too: from the same logic program there are more transitions generated using general semantics than synchronous (non-deterministic) than asynchronous semantics. The learning time is rather similar for all semantics when the number of transitions is the same. This was quite expected since it only impacts the pre-processing phase of the algorithm when searching for negative examples of each variable values.

Semantics	Mammalian (10)	Fission (10)	Budding (12)	Arabidopsis (15)
Synchronous	1.84s / 1,024	1.55s / 1,024	34.48s / 4,096	2,066s / 32,768
Asynchronous	19.88s / 4,273	19.18s / 4,217	523s / 19,876	T.O. / 213,127
General	928s / 34,487	1,220s / 29,753	T.O. / 261,366	T.O. / > 500,000

Table 1: Run time of **GULA** (run time in seconds / number of transitions as input) for Boolean network benchmarks up to 15 nodes for the three semantics.

Table 1 shows the run time of **GULA** when learning Boolean networks from [4]. Here we reproduced the experiments held in [13]: all transitions of each benchmark are generated but only for the three considered semantics. The table also provides the number of transitions generated by the semantics for each benchmark. It is important to note that those systems are all synchronous deterministic, meaning that in the synchronous case, the input transitions of **GULA** are the same as the input transitions of **LF1T** in [13]. Here the number of transitions in the synchronous case is much lower than for the random experiment, which explains the difference in terms of run-time. Furthermore the rules to be learned are quite simpler in the sense that the number of conditions (i.e., the size of the body of all rules) never exceed 6 and the total number of rules is always below 124. Nevertheless, the new algorithm is slower than **LF1T** [13] in the synchronous deterministic Boolean case, which is expected since it is not specifically dedicated to learning such networks: **GULA** learns both values (0 and 1) of each variable and pre-processes the transitions before learning rules, while **LF1T** is optimized to only learn rules that make a variable take the value 1 in the next state. On the other hand, asynchronous and general semantics transitions can only be handled by our new algorithm.

GULA succeeds to learn the two smaller benchmarks for all semantics in less than an hour. The 12 variables benchmark could be learned for both the synchronous and asynchronous semantics. For the general semantics, however, it took more that the 10 hours time-out (T.O. in the table), which is due to the very high number of transitions generated by those semantics: more than 200,000. The 15 variables benchmark could only be learned in synchronous case,

both asynchronous and general semantics cases having reached the time-out. The current implementation of the algorithm is rather naive and better performances are expected from future optimizations. In particular, the algorithm can be parallelized into as many threads as there are different rule heads (one thread per variable value). We are also developping an approximated version of the algorithm based on local search which could be free of the combinatorial explosion.

5 Conclusions

In this paper we proposed a modeling of multi-valued system in the form of annotated logic programs. A modeling of both synchronous, asynchronous and general semantics is also proposed. From this solid theory a rather straightforward extension of the **LF1T** algorithm of [13] is proposed. **LF1T** is restricted to learning synchronous deterministic Boolean systems. **GULA** can capture multi-valued deterministic or non-deterministic systems from transitions produced by synchronous, asynchronous and general semantics. The current implementation of the algorithm is rather naive which can explain the quite poor performances compared to previous algorithms on the same ground. Scalability can be greatly improved through practical optimization as well as the design of heuristic methods based on local search, which are our current focus.

This work opens the way to several theoretical extensions, among them: taking into account dead-ends, memory or update delays; tackling incomplete or incoherent dynamics; formally characterizing the semantics that can, or cannot, be learned; learning the semantics alongside the dynamical rules. Once answered, each of these subjects can lead to the creation of a new level of learning greatly extending the expressive power of the **LFIT** framework. A long-term objective is to fully automate the learning of models directly from time series, that is, gene expression measurements during time, the semantics of which is unknown or even changeable.

Other possible practical or technical extensions include optimization additions, such as a massive parallelization of the learning of each head of rule, or the inclusion of heuristics to guide the learning.

References

1. H. A. Blair and V. Subrahmanian. Paraconsistent foundations for logic programming. *Journal of non-classical logic*, 5(2):45–73, 1988.
2. H. A. Blair and V. Subrahmanian. Paraconsistent logic programming. *Theoretical Computer Science*, 68(2):135 – 154, 1989.
3. T. Chatain, S. Haar, and L. Paulevé. Boolean networks: Beyond generalized asynchronicity. In *AUTOMATA 2018*. Springer, 2018.
4. E. Dubrova and M. Teslenko. A SAT-based algorithm for finding attractors in synchronous boolean networks. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 8(5):1393–1399, 2011.

5. M. Fitting. Bilattices and the semantics of logic programming. *The Journal of Logic Programming*, 11(2):91 – 116, 1991.
6. A. Garg, A. Di Cara, I. Xenarios, L. Mendoza, and G. De Micheli. Synchronous versus asynchronous modeling of gene regulatory networks. *Bioinformatics*, 24(17):1917–1925, 2008.
7. M. L. Ginsberg. Multivalued logics: A uniform approach to reasoning in artificial intelligence. *Computational intelligence*, 4(3):265–316, 1988.
8. K. Inoue. Logic programming for boolean networks. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 924, 2011.
9. K. Inoue, T. Ribeiro, and C. Sakama. Learning from interpretation transition. *Machine Learning*, 94(1):51–79, 2014.
10. M. Kifer and V. Subrahmanian. Theory of generalized annotated logic programming and its applications. *Journal of Logic Programming*, 12(4):335–367, 1992.
11. D. Martınez, G. Alenya, C. Torras, T. Ribeiro, and K. Inoue. Learning relational dynamics of stochastic domains for planning. In *Proceedings of the 26th International Conference on Automated Planning and Scheduling*, 2016.
12. D. Martínez Martínez, T. Ribeiro, K. Inoue, G. Alenyà Ribas, and C. Torras. Learning probabilistic action models from interpretation transitions. In *Proceedings of the Technical Communications of the 31st International Conference on Logic Programming (ICLP 2015)*, pages 1–14, 2015.
13. T. Ribeiro and K. Inoue. Learning prime implicant conditions from interpretation transition. In *Inductive Logic Programming*, pages 108–125. Springer, 2015.
14. T. Ribeiro, M. Magnin, K. Inoue, and C. Sakama. Learning delayed influences of biological systems. *Frontiers in Bioengineering and Biotechnology*, 2:81, 2015.
15. T. Ribeiro, M. Magnin, K. Inoue, and C. Sakama. Learning multi-valued biological models with delayed influence from time-series observations. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pages 25–31, Dec 2015.
16. T. Ribeiro, S. Tourret, M. Folschette, M. Magnin, D. Borzacchiello, F. Chinesta, O. Roux, and K. Inoue. Inductive learning from state transitions over continuous domains. In N. Lachiche and C. Vrain, editors, *Inductive Logic Programming*, pages 124–139, Cham, 2018. Springer International Publishing.
17. M. H. Van Emden. Quantitative deduction and its fixpoint theory. *The Journal of Logic Programming*, 3(1):37–53, 1986.

A Appendix: proofs of Section 2.1

Proposition 10 (Prop. 1). *Let R_1, R_2 be two $\mathcal{M}$VL rules. If $R_1 \leq R_2$ and $R_2 \leq R_1$ then $R_1 = R_2$.*

Proof. Let R_1, R_2 be two $\mathcal{M}$VL rules such that $R_1 \leq R_2$ and $R_2 \leq R_1$. Then $h(R_1) = h(R_2)$ and $b(R_1) \subseteq b(R_2)$ and $b(R_2) \subseteq b(R_1)$, hence $b(R_1) \subseteq b(R_2) \subseteq b(R_1)$ thus $b(R_1) = b(R_2)$ and $R_1 = R_2$. □

Proposition 11 (Prop. 2: Cross-matching). *Let R and R' be two $\mathcal{M}$VL rules.*

$$R \sqcap R' \text{ iff } \forall \mathrm{v} \in \mathcal{V}, \forall val, val' \in \mathbb{N}, (\mathrm{v}^{val}, \mathrm{v}^{val'}) \in b(R) \times b(R') \implies val = val'.$$

Proof. For the direct implication, assume given two $\mathcal{M}$VL rules R and R' such that $R \sqcap R'$. By definition, there exists $s \in \mathcal{S}$ such that $R \sqcap s$ and $R' \sqcap s$. Also by definition, for all $(\mathrm{v}^{val}, \mathrm{v}^{val'}) \in b(R) \times b(R')$, there exists $\mathrm{v}^{val}, \mathrm{v}^{val'} \in s$. Moreover, by the definition of a state, $\mathrm{v}^{val} = \mathrm{v}^{val'}$, thus $val = val'$.

For the reverse implication, consider a state s so that $b(R) \cup b(R') \subseteq s$. This is compatible with the definition of a state because if a variable $\mathrm{v} \in \mathcal{V}$ is featured in both $b(R)$ and $b(R')$, that is, if there exists $val, val' \in \mathbb{N}$ so that $(\mathrm{v}^{val}, \mathrm{v}^{val'}) \in b(R) \times b(R')$, then $val = val'$ and $\mathrm{v}^{val} = \mathrm{v}^{val'}$. For variables not featured in both $b(R)$ and $b(R')$, we can chose any value in the domain of the variable. As a consequence, we have: $b(R) \subseteq s$ and $b(R') \subseteq s$, which gives: $R \sqcap s$ and $R' \sqcap s$, meaning: $R \sqcap R'$. □

B Appendix: proofs of Section 2.2

Theorem 7 (Th. 1: properties of the least revision). *Let R be a $\mathcal{M}$VL rule and $s \in \mathcal{S}$ such that $R \sqcap s$. Let $S_R := \{s' \in \mathcal{S} \mid R \sqcap s'\}$ and $S_{\text{spe}} := \{s' \in \mathcal{S} \mid \exists R' \in L_{\text{spe}}(R, s), R' \sqcap s'\}$.*

Let P be a $\mathcal{M}$VLP and $T, T' \subseteq \mathcal{S}^2$ such that $|\text{fst}(T)| = 1 \wedge \text{fst}(T) \cap \text{fst}(T') = \emptyset$. The following results hold:

1. $S_{\text{spe}} = S_R \setminus \{s\}$,
2. $L_{\text{rev}}(P, T)$ *is consistent with* T,
3. $\overset{P}{\hookrightarrow} T' \implies \overset{L_{\text{rev}}(P,T)}{\hookrightarrow} T'$,
4. $\overset{P}{\hookrightarrow} T \implies \overset{L_{\text{rev}}(P,T)}{\hookrightarrow} T$,
5. P *is complete* $\implies L_{\text{rev}}(P, T)$ *is complete.*

Proof.

1. First, let us suppose that $\exists s'' \notin S_R \setminus \{s\}$ such that $\exists R' \in L_{\text{spe}}(R, s), R' \sqcap s''$. By definition of matching $R' \sqcap s'' \implies b(R') \subseteq s''$. By definition of least specialization, $b(R') = b(R) \cup \{\mathrm{v}^{val}\}, \mathrm{v}^{val'} \in s, \mathrm{v}^{val''} \notin b(R), val \neq val'$. Let us suppose that $s'' = s$, then $b(R') \not\subseteq s''$ since $\mathrm{v}^{val} \in b(R')$ and $\mathrm{v}^{val} \notin s$, this

is a contradiction. Let us suppose that $s'' \neq s$ then $\neg R \sqcap s''$, thus $b(R) \not\subseteq s''$ and $b(R') \not\subseteq s''$, this is a contradiction.
Second, let us assume that $\exists s'' \in S_R \setminus \{s\}$ such that $\forall R' \in L_{\text{spe}}(R,s), \neg R' \sqcap s''$. By definition of S_R, $R \sqcap s''$. By definition of matching $\neg R' \sqcap s'' \implies b(R') \not\subseteq s''$. By definition of least specialization, $b(R') = b(R) \cup \{\mathrm{v}^{val}\}, \mathrm{v}^{val'} \in s, val \neq val'$. By definition of matching $R \sqcap s'' \implies b(R) \subseteq s'' \implies s'' = b(R) \cup I, b(R) \cap I = \emptyset$ and thus $b(R') \not\subseteq s'' \implies \mathrm{v}^{val} \notin I$. The assumption implies that $\forall \mathrm{v}^{val'} \in I, \forall R' \in L_{\text{spe}}(R,s), \mathrm{v}^{val} \in b(R'), val \neq val'$. By definition of least specialization, it implies that $\mathrm{v}^{val'} \in s$ and thus $I = s \setminus b(R)$ making $s'' = s$, which is a contradiction.
Conclusion: $S_{\text{spe}} = S_R \setminus \{s\}$

2. By definition of a consistent program, if two sets of $\mathcal{M}$VL rules SR_1, SR_2 are consistent with T then $SR_1 \cup SR_2$ is consistent with T. Let $R_P = \{R \in P \mid R \sqcap s, \forall (s,s') \in T, h(R) \not\sqsubseteq s'\}$ be the set of rules of P that conflict with T. By definition of least revision $L_{\text{rev}}(P,T) = (P \setminus R_P) \cup \bigcup_{R \in R_P} L_{\text{spe}}(R,s)$. The first part of the expression $P \setminus R_P$ is consistent with T since $\nexists R' \in P \setminus R_P$ such that R' conflicts with T. The second part of the expression $\bigcup_{R \in R_P} L_{\text{spe}}(R,s)$ is also consistent with T: $\nexists R' \in L_{\text{spe}}(R,s), R' \sqcap s$ thus $\nexists R' \in L_{\text{spe}}(R,s)$ that conflict with T and $\bigcup_{R \in R_P} L_{\text{spe}}(R,s)$ is consistent with T. Conclusion: $L_{\text{rev}}(P,T)$ is consistent with T.
3. Let $(s_1, s_2) \in T'$ thus $s_1 \neq s$. From definition of realization, $\mathrm{v}^{val} \in s_2 \implies \exists R \in P, h(R) = \mathrm{v}^{val}, R \sqcap s_1$. If $\neg R \sqcap s$ then $R \in L_{\text{rev}}(P,T)$ and $\xrightarrow{L_{\text{rev}}(P,T)} (s_1, s_2)$. If $R \sqcap s$, from the first point $\exists R' \in L_{\text{spe}}(R,s), R' \sqcap s_1$ and since $h(R') = h(R) = \mathrm{v}^{val}, \xrightarrow{L_{\text{rev}}(P,T)} (s_1, s_2)$. Applying this reasoning on all elements of T' implies that $\xhookrightarrow{P} T' \implies \xrightarrow{L_{\text{rev}}(P,T)} T'$.
4. Let $(s_1, s_2) \in T$, since $\xhookrightarrow{P} T$ by definition of realization $\forall \mathrm{v}^{val} \in s_2, \exists R \in P, R \sqcap s_1, h(R) = \mathrm{v}^{val}$. By definition of conflict, R is not in conflict with T thus $R \in L_{\text{rev}}(P,T)$ and $\xrightarrow{L_{\text{rev}}(P,T)} T$.
5. Let $(s_1, s_2) \in \mathcal{S}^2$, if P is complete, then by definition of a complete program $\forall \mathrm{v} \in \mathcal{V}, \exists R \in P, R \sqcap s_1, \text{var}(h(R)) = v$. If $\neg R \sqcap s$ then $R \in L_{\text{rev}}(P,T)$. If $R \sqcap s$, from the first point $\exists R' \in L_{\text{spe}}(R,s), R' \sqcap s_1$ and thus $R' \in L_{\text{rev}}(P,T)$ and since $\text{var}(h(R')) = \text{var}(h(R)) = \mathrm{v}$, $L_{\text{rev}}(P,T)$ is complete.

□

Proposition 12 (Prop. 4: optimal program of empty set). $P_{\mathcal{O}}(\emptyset) = \{\mathrm{v}^{val} \leftarrow \emptyset \mid \mathrm{v}^{val} \in \mathcal{A}\}$.

Proof. Let $P = \{\mathrm{v}^{val} \leftarrow \emptyset \mid \mathrm{v}^{val} \in \mathcal{A}\}$. The $\mathcal{M}$VLP P is consistent and complete by construction. Like all $\mathcal{M}$VLPs, $\xhookrightarrow{P} \emptyset$ and there is no transition in $\emptyset$ to match with the rules in P. In addition, by construction, the rules of P dominate all $\mathcal{M}$VL rules. □

Proposition 13 (Prop. 3: uniqueness of optimal program). *Let $T \subseteq \mathcal{S}^2$. The $\mathcal{M}$*VLP *optimal for T is unique and denoted $P_{\mathcal{O}}(T)$.*

Proof. Let $T \subseteq \mathcal{S}^2$. Assume the existence of two distinct $\mathcal{M}$VLPs optimal for T, denoted by $P_{\mathcal{O}_1}(T)$ and $P_{\mathcal{O}_2}(T)$ respectively. Then w.l.o.g. we consider that there exists a $\mathcal{M}$VL rule R such that $R \in P_{\mathcal{O}_1}(T)$ and $R \notin P_{\mathcal{O}_2}(T)$. By the definition of a suitable program, R is not conflicting with T and there exists a $\mathcal{M}$VL rule $R_2 \in P_{\mathcal{O}_2}(T)$, such that $R \leq R_2$. Using the same definition, there exists $R_1 \in P_{\mathcal{O}_1}(T)$ such that $R_2 \leq R_1$ since R_2 is not conflicting with T. Thus $R \leq R_1$ and by the definition of an optimal program $R_1 \leq R$. By Proposition 1, $R_1 = R$ and thus $R \leq R_2 \leq R$ hence $R_2 = R$, a contradiction. □

Theorem 8 (Th. 2: least revision is suitable). *Let $s \in \mathcal{S}$ and $T, T' \subseteq \mathcal{S}^2$ such that $|\text{fst}(T')| = 1 \wedge \text{fst}(T) \cap \text{fst}(T') = \emptyset$. $L_{\text{rev}}(P_{\mathcal{O}}(T), T')$ is a $\mathcal{M}$*VLP *suitable for $T \cup T'$.*

Proof. Let $P = L_{\text{rev}}(P_{\mathcal{O}}(T), T')$. Since $P_{\mathcal{O}}(T)$ is consistent with T, by Theorem 1, P is also consistent with T and thus consistent with $T' \cup T$. Since $P_{\mathcal{O}}(T)$ realize T by Theorem 1, $\overset{P}{\hookrightarrow} T$. Since $s \notin \text{fst}(T)$, a $\mathcal{M}$VL rule R such that $b(R) = s$ does not conflict with T. By definition of suitable program $\exists R' \in P_{\mathcal{O}}(T), R \leq R'$, thus $\overset{P_{\mathcal{O}}(T)}{\hookrightarrow} T'$. Since $\overset{P_{\mathcal{O}}(T)}{\hookrightarrow} T'$ by Theorem 1 $\overset{P}{\hookrightarrow} T'$ and thus $\overset{P}{\hookrightarrow} T \cup T'$. Since $P_{\mathcal{O}}(T)$ is complete, by Theorem 1, P is also complete. To prove that P verifies the last point of the definition of a suitable $\mathcal{M}$VLP, let R be a $\mathcal{M}$VL rule not conflicting with $T \cup T'$. Since R is also not conflicting with T, there exists $R' \in P_{\mathcal{O}}(T)$ such that $R \leq R'$. If R' is not conflicting with T', then R' will not be revised and $R' \in P$, thus R is dominated by a rule of P. Otherwise, R' is in conflict with T', thus $R' \sqcap s$ and $\forall (s, s') \in T', h(R') \notin s'$. Since R is not in conflict with T' and $h(R) = h(R')$, since $R \leq R'$ then $b(R) = b(R') \cup I, \exists \text{v}^{val} \in I, \text{v}^{val} \notin s$. By definition of least revision and least specialization, there is a rule $R'' \in L_{\text{spe}}(R', s)$ such that $\text{v}^{val} \in b(R'')$ and since $R'' = h(R') \leftarrow b(R') \cup \text{v}^{val}$ thus $R \leq R''$. Thus R is dominated by a rule of P. □

C Appendix: proofs of Section 2.3

Proposition 14 (Prop. 6 Synchronous transitions). *Let $T \subseteq \mathcal{S}^2$ so that* $\text{fst}(T) = \mathcal{S}$. *The transitions of T are* synchronous*, i.e., $\exists P$ a $\mathcal{M}$*VLP *such that $\mathcal{T}_{syn}(P) = T$, if and only if $\forall (s, s_1), (s, s_2) \in T, \forall s_3 \in \mathcal{S}, s_3 \subseteq s_1 \cup s_2 \implies (s, s_3) \in T$.*

Proof. ($\Rightarrow$) Let (s, s_1) and (s, s_2) in T and $s_3 \in \mathcal{S}$ so that $s_3 \subseteq s_1 \cup s_2$. Let $A := \{h(R) \mid R \in P, R \sqcap s_\alpha\}$. Then it comes: $s_1, s_2 \subseteq A$, thus: $s_3 \subseteq (s_1 \cup s_2) \subseteq A$. By construction of $\mathcal{T}_{syn}$, it comes: $(s, s_3) \in T$.

($\Leftarrow$) Consider $P := \{\text{v}^{val} \leftarrow s \mid (s, s') \in T \wedge \text{v}^{val} \in s'\}$ the program made of the most specific rules that realize T.

($\subseteq$) Let $(s, s') \in \mathcal{T}_{syn}(P)$ and let $x \in s'$. By construction of P, there is a rule R such that $h(R) = x$. Therefore, there exists a state $s^x \in \mathcal{S}$ such that

$x \in s^x$ and $(s, s^x) \in T$. This reasoning can be carried for all atoms x in s', and in the end: $s' \subseteq \bigcup_{x \in s'} s^x$. By applying the proposition for each x, it comes: $(s, s') \in T$.

($\supseteq$) Let $(s, s') \in T$. By construction of P, we have $s' \subseteq \{h(R) \mid R \in P, R \sqcap s\}$. Thus, $(s, s') \in \mathcal{T}_{syn}(P)$. □

Proposition 15 (Prop. 7 Asynchronous transitions). *Let $T \subseteq \mathcal{S}^2$ so that* $\mathrm{fst}(T) = \mathcal{S}$. *The transitions of T are* asynchronous, *i.e., $\exists P$ a $\mathcal{M}$*VLP *such that $\mathcal{T}_{asyn}(P) = T$, if and only if $\forall s, s' \in \mathcal{S}, s \neq s', \big((s, s) \in T \implies (s, s') \notin T\big) \wedge \big((s, s') \in T \implies |s \setminus s'| = 1\big)$.*

Proof. ($\Rightarrow$) Let $s, s' \in \mathcal{S}$ so that $s \neq s'$. • First suppose that $(s, s) \in T$. Then, by construction of $\mathcal{T}_{asyn}(P)$, $\forall R \in P, R \sqcap s \implies h(R) \in s$. If $(s, s') \in T$, then $\exists R \in P, R \sqcap s \wedge h(R) \notin s$, which is a contradiction. • Now suppose that $(s, s') \in T$. Then, still by construction of $\mathcal{T}_{asyn}(P)$, there exists $R \in P$ so that $s' = s \backslash\backslash \{h(R)\}$ and $h(R) \notin s$. Thus, $|s \setminus s'| = 1$.

($\Leftarrow$) Consider $P := \{\mathrm{v}^{val} \leftarrow s \mid (s, s') \in T \wedge \mathrm{v}^{val} \in s'\}$ the program made of the most specific rules that realize T.

($\subseteq$) Let $(s, s') \in \mathcal{T}_{asyn}(P)$. • First suppose that $s = s'$. By construction of $\mathcal{T}_{asyn}$, it means that $\forall R \in P, R \sqcap s \implies h(R) \in s$ which, by construction of P, means that $(s, s) \in T$. • Now suppose that $s \neq s'$. By construction of $\mathcal{T}_{asyn}$, it means that there exists $R \in P$ so that $h(R) \notin s$ and $s' = s \backslash\backslash \{h(R)\}$. By construction of P, this means that there exists $(s, s'') \in T$ so that $h(R) \in s''$ (and thus $s'' \neq s$). Moreover, from the right-hand part of the property, it comes that $|s \setminus s''| = 1$, that is, there exists a unique $x \in \mathcal{A}$ such that $x \notin s$ and $s'' = s \backslash\backslash \{x\}$. Necessarily, $x = h(R)$, thus $s'' = s'$ and $(s, s') \in T$.

($\supseteq$) Let $(s, s') \in T$. • First suppose that $s = s'$. From the left-hand side of the property, there exists no transition $(s, s'') \in T$ such that $s'' \neq s$. By construction, it comes: $(s, s) \in \mathcal{T}_{asyn}(P)$. • Now suppose that $s \neq s'$. From the right-hand side of the property, $|s \setminus s''| = 1$, meaning that there exists $x \in \mathcal{A}$ such that $s' = s \backslash\backslash x$ and $x \notin s$. Thus there exists a rule $R \in P$ such that $b(R) = s$ and $h(R) \in (s' \setminus s)$. As $(s' \setminus s) = \{x\}$, and by construction, we have: $(s, s') \in \mathcal{T}_{asyn}(P)$. □

Proposition 16 (Prop. 8 General transitions). *Let $T \subseteq \mathcal{S}^2$ so that* $\mathrm{fst}(T) = \mathcal{S}$. *The transitions of T are* general, *i.e., $\exists P$ a $\mathcal{M}$*VLP *such that $\mathcal{T}_{gen}(P) = T$, if and only if: $\forall (s, s_1), (s, s_2) \in T, \forall s_3 \in \mathcal{S}, s_3 \subseteq s \cup s_1 \cup s_2 \implies (s, s_3) \in T$.*

Proof. ($\Rightarrow$) Let (s, s_1) and (s, s_2) in T and $s_3 \in \mathcal{S}$ so that $s_3 \subseteq s \cup s_1 \cup s_2$. Let $A := \{h(R) \mid R \in P, R \sqcap s_\alpha\}$. Then it comes: $s_1, s_2 \subseteq A$, thus: $(s_3 \setminus s) \subseteq (s_1 \cup s_2) \subseteq A$. By construction of $\mathcal{T}_{gen}$, it comes: $(s, s \backslash\backslash (s_3 \setminus s)) = (s, s_3) \in T$.

($\Leftarrow$) Consider $P := \{\mathrm{v}^{val} \leftarrow s \mid (s, s') \in T \wedge \mathrm{v}^{val} \in s'\}$ the program made of the most specific rules that realize T.

($\subseteq$) Let $(s, s') \in \mathcal{T}_{gen}(P)$ and let $x \in s'$. By definition of $\mathcal{T}_{gen}$, either $x \in s$ or there exists $R \in P$ so that $x = h(R)$. In the first case, let $s^x = s$; in the second case, by construction of P, there exists a state $s^x \in T$ so that $(s, s^x) \in T$ and $x \in s^x$. By carrying this reasoning for all atoms x in s', we have: $s' \subseteq \bigcup_{x \in s'} s^x$

where for some x, we have $s^x = s$. By applying the proposition for each x, it comes: $(s, s') \in T$.

($\subseteq$) Let $(s, s') \in T$. By construction of P, $s' \subseteq \{h(R) \mid R \in P \wedge R \sqcap s\}$. Thus, $(s, s \backslash\backslash s') = (s, s') \in \mathcal{T}_{gen}(P)$. □

Theorem 9 (Semantics-free correctness). *Let P be a $\mathcal{M}$VLP such that P is complete.*

- $\mathcal{T}_{syn}(P) = \mathcal{T}_{syn}(P_{\mathcal{O}}(\mathcal{T}_{syn}(P)))$,
- $\mathcal{T}_{asyn}(P) = \mathcal{T}_{asyn}(P_{\mathcal{O}}(\mathcal{T}_{asyn}(P)))$,
- $\mathcal{T}_{gen}(P) = \mathcal{T}_{gen}(P_{\mathcal{O}}(\mathcal{T}_{gen}(P)))$.

Proof. Let us first consider the case of $\mathcal{T}_{syn}$. Let $T \subseteq \mathcal{S}^2$ so that $\text{fst}(T) = \mathcal{S}$ and let $M(P, s)$ be the set of heads of rules of P that match the state s: $M(P, s) := \{h(R) \mid R \in P \wedge R \sqcap s\}$.

We first expose two properties about $M(P_{\mathcal{O}}(T), s)$. According to Def. 12, $P_{\mathcal{O}}(T)$ realizes T, thus: (a) $\forall (s, s') \in T, s' \subseteq M(P_{\mathcal{O}}(T), s)$. According to the same definition, $P_{\mathcal{O}}(T)$ is consistent with T, thus: (b) $\forall s \in \mathcal{S}, \forall \mathrm{v}^{val} \in M(P_{\mathcal{O}}(T), s), \exists (s, s') \in T, \mathrm{v}^{val} \in s'$.

Now we prove by contradiction that $\mathcal{T}_{syn}(P) = \mathcal{T}_{syn}(P_{\mathcal{O}}(\mathcal{T}_{syn}(P)))$. Thus, suppose $\mathcal{T}_{syn}(P) \neq \mathcal{T}_{syn}(P_{\mathcal{O}}(\mathcal{T}_{syn}(P)))$ and let $T := \mathcal{T}_{syn}(P)$. Therefore, there exists $s \in \mathcal{S}$ so that $M(P, s) \neq M(P_{\mathcal{O}}(T), s)$. Thus:

- Either $\exists \mathrm{v}^{val} \in M(P, s), \mathrm{v}^{val} \notin M(P_{\mathcal{O}}(T), s)$ and thus $\exists (s, s') \in T, s' \not\subseteq M(P_{\mathcal{O}}(T), s)$ which is a contradiction with (a).
- Or $\exists \mathrm{v}^{val} \in M(P_{\mathcal{O}}(T), s), \mathrm{v}^{val} \notin M(P, s)$ and thus $\nexists (s, s') \in T, \mathrm{v}^{val} \in s'$ which is a contradiction with (b).

Then $\forall s \in \mathcal{S}, M(P, s) = M(P_{\mathcal{O}}(T), s)$ and according to Def. 14, $\mathcal{T}_{syn}(P) = \mathcal{T}_{syn}(P_{\mathcal{O}}(\mathcal{T}_{syn}(P)))$.

The case of $\mathcal{T}_{gen}$ is similar with $M(P, s) := s \cup \{h(R) \mid R \in P \wedge R \sqcap s\}$ and noting that $\mathcal{T}_{gen}(P) = \{(s, s') \in \mathcal{S}^2 \mid s' \subseteq M(P, s)\}$. The case of $\mathcal{T}_{asyn}$ is also similar with $M(P, s) := \{h(R) \mid R \in P \wedge R \sqcap s\} \setminus s$ and noting that $\mathcal{T}_{asyn}(P) = \{(s, s \backslash\backslash x) \in \mathcal{S}^2 \mid x \in M(P, s)\} \cup \{(s, s) \in \mathcal{S}^2 \mid M(P, s) = \emptyset\}$. □

D Appendix: proofs of Section 3

Theorem 10 (Th. 10: GULA Termination, soundness, completeness, optimality). *Let $T \subseteq \mathcal{S}^2$. The call* **GULA**$(\mathcal{A}, T)$ *terminates and* **GULA**$(\mathcal{A}, T) = P_{\mathcal{O}}(T)$.

Proof. Let $T \subseteq \mathcal{S}^2$. The call **GULA**$(T)$ terminates because all loops iterate on finite sets.

The algorithm iterates over atom v^{val} iteratively to extract all state s such that $(s, s') \in T \implies \mathrm{v}^{val} \notin s'$. This is equivalent to generate the set $TT = \{T' \subseteq T \mid \forall t, t' \in T \implies t = (s, s'), t' = (s, s'')\}$.

To prove that **GULA**(T)=$P_{\mathcal{O}}(T)$, and is thus sound, complete and optimal, it suffices to prove that the main loop (Algorithm 1, lines 16–32) preserves the invariant $P = P_{\mathcal{O}}(T_i)$ after the i^{th} iteration where T_i is the union of all set of transitions already selected line 16 after the i^{th} iteration for all i from 0 to $|TT|$.

Line 15 initializes P to $\{\mathrm{v}^{val} \leftarrow \emptyset\}$. Thus by Proposition 4, after line 15, $P = \{R \in P_{\mathcal{O}}(\emptyset) \mid h(R) = \mathrm{v}^{val}\}$.

Let us assume that before the $(i+1)^{\text{th}}$ iteration of the main loop, $P = \{R \in P_{\mathcal{O}}(T_i) \mid h(R) = \mathrm{v}^{val}\}$. Through the loop of lines 18–20, $P' = \{R \in P_{\mathcal{O}}(T_i) \mid R \text{ does not conflict with } T_{i+1}, h(R) = \mathrm{v}^{val}\}$ is computed. Then the set $P'' = \bigcup\{L_{\text{spe}}(R, s) \mid R \in P_{\mathcal{O}}(T_i) \backslash P', h(R) = \mathrm{v}^{val}\}$ is iteratively build through the calls to **least_specialization** at line 22 and the dominated rules are pruned as they are detected by the loop of lines 23–32. Thus by Theorem 2 and Proposition 5, $P = \{R \in P_{\mathcal{O}}(T_{i+1}) \mid h(R) = \mathrm{v}^{val}\}$ after the $(i+1)^{\text{th}}$ iteration of the main loop. Since the same operation is hold for each $\mathrm{v}^{val} \in \mathcal{A}$, $P = \bigcup\{R \in P_{\mathcal{O}}(\bigcup TT = T) \mid h(R) = \mathrm{v}^{val}\} = P_{\mathcal{O}}(T)$ after all iterations of the loop of line line 3. □

Theorem 11 (GULA Complexity). *Let $T \subseteq \mathcal{S}^2$ be a set of transitions, $n := |\mathcal{V}|$ be the number of variables of the system and $d := \max(\mathsf{dom}(\mathcal{V}))$ be the maximal number of values of its variables. The worst-case time complexity of* **GULA** *when learning from T belongs to $\mathcal{O}(|T|^2 + 2n^3d^{2n+1} + 2n^2d^n)$ and its worst-case memory use belongs to $\mathcal{O}(d^{2n} + 2d^n + nd^{n+2})$.*

Proof. The algorithm takes as input a set of transition $T \subseteq \mathcal{S}^2$ bounding the memory use to $O(|\mathcal{S}^2|) = O(d^n \times d^n) = O(d^{2n})$ at start. The learning is performed iteratively for each possible rule head $\mathrm{v}^{val} \in \mathcal{A}$. The extraction of negative example requires to compare each transition of T one to one and thus has a complexity of $op_1 = O(|T|^2)$. Those transitions are stored in $Neg_{\mathrm{v}^{val}}$ which size is at most $|\mathcal{S}|$ extending the memory use to $O(d^{2n} + d^n)$.

The learning phase revises a set of rule $P_{\mathrm{v}^{val}}$ where each rule has the same head v^{val}. There are at most d^n possible rule bodies and thus $|P_{\mathrm{v}^{val}}| \leq d^n$, the memory use of $|P_{\mathrm{v}^{val}}|$ is then $O(d^n)$ extending the memory bound to $O(d^{2n} + d^n + d^n) = O(d^{2n} + 2d^n)$.

For each state s of $Neg_{\mathrm{v}^{val}}$, each rule of $P_{\mathrm{v}^{val}}$ that matches s are extracted into a set of rules R_m. This operation has a complexity of $op_2 = O(d^n \times n^2)$. Each rule of R_m are then revised using least specialization, this operation has a complexity of $O(n^2)$. $|R_m| \leq d^n$ thus the revision of all matching rules is $op_3 = O(d^n \times n^2)$. All revision are stored in LS and there are at most dn revisions for each rule, thus $|LS| \leq d^n \times dn$ extending the memory bound to $O(d^{2n} + 2d^n + nd^{n+1}) = O(d^{2n} + 2d^n + n \times d^{n+2})$.

The memory usage of **GULA** is therefore $O(d^{2n} + 2d^n + n \times d^{n+2})$.

All rules of LS are compared to the rule of $P_{\mathrm{v}^{val}}$ for domination check, this operation has a complexity of $op_4 = O(2 \times |LS| \times |P_{\mathrm{v}^{val}}| \times n^2) = O(2 \times d^n \times d \times n \times d^n \times n^2) = O(2 \times n^3 \times d^{2n+1})$. The complexity is bound by $O(op_1 + op_2 + op_3 + op_4) = O(|T|^2 + d^n \times n^2 + d^n \times n^2 + 2 \times n^3 \times d^{2n+1}) = O(|T|^2 + 2 \times n^3 \times d^{2n+1} + 2 \times n^2 d^n)$

The computational complexity of **GULA** is thus $O(|T|^2 + 2n^3d^{2n+1} + 2n^2d^n)$. □